**CS2106 Introduction to Operating Systems**

**Lab 4**

**Contiguous Memory Allocation**

**Answer Book**

**Submission checklist:** A ZIP file called AxxxxxxY.zip, where AxxxxxxY is the student ID of the student submitting. The ZIP file should contain:

* Your answer book, properly renamed.
* Your malloc.c for ff, nf, bf, wf

|  |  |
| --- | --- |
| **Student 1** | |
| Name: | Kenneth Seet |
| Student ID (AxxxxxxY): | A0258173Y |
| Group (Bxx): | B18 |
| **Student 2** | |
| Name: | Daniel Wang |
| Student ID (AxxxxxxY): | A0255689H |
| Group (Bxx): | B11 |

Question 3.1 (1 mark)

Our bitmap would be 8 bytes in size.

Question 3.2 (1 mark)

It would not make a difference if the array is of type unsigned char, as the characters are only used for bit operations and not arithmetic. In addition, the memory used by both types are the same, so even though char represents values from -128 to 127 and unsigned char represents values from 0 to 255, there are no differences in the context of this use case.

Question 3.3 (1 mark)

Our myfree routine needs to know how many bytes of memory need to be freed.

We initialised the TData struct, setting the len attribute of TData to the number of bytes that needed to be freed.

We then initialised the TNode struct with the key being the starting index of the memory allocation, and the pdata as the TData struct we initialised earlier.

When myfree is called, we obtain the starting index of the memory allocation using get\_index(), call find\_node(\_memlist, index), where the index is the one obtained from get\_index(), and obtain the len attribute from the pdata attribute of the obtained node.

We then call free\_map using the len attribute obtained earlier.

Question 4.1 (1 mark)

*int* isAllocated; // Boolean to check if memory is allocated  
*int* len; // Length of memory segment

Question 4.2 (1 mark)

size of TData: 2 int x 4B = 8B

size of TNode: 5 ptr x 8B + 1 int x 4B = 44B

size of ptr: 8B

size of int: 4B

Best Case: In the best case scenario, there is only one empty TNode, with TData. Since the size of a TNode is 44 bytes and the size of TData is 8 bytes, the storage requirement for the linked list in the best case is 52 bytes.

Worst Case: In the worst case scenario, the entire heap (64KB) is filled with TNode structures, each with one TData. Assuming each TNode takes 44 bytes and each TData takes up 8 bytes, and there are 64 \* 1024 (64KB) nodes, the total storage requirement would be 64 \* 1024 \* (44 + 8) bytes, which equals 3407872 bytes.

Question 4.3 (1 mark)

The Next-fit algorithm requires us to implement a TNode\*, which is a pointer to the last allocated block of memory. This pointer keeps track of the last allocated block, and it is updated every time an allocation is performed.

This additional information is necessary because Next-Fit starts searching for a free block from the location where the previous allocation ended rather than always starting from the beginning of the memory list like in First-Fit.

Question 4.4 (1 mark)

Since the amount of space we are using is fixed, unlike in a linked list, the best and worst case space requirements are identical.

Number of memory units = 64 KB / 1B = 2^16

(2^16) / (2^3) = 8192B

Both best and worst case requirements: 8192 bytes

Question 4.5 (1 mark)

In question 4.2, when using a linked list, the best and worst case scenario vary drastically, with the minimal overhead being 52 bytes and the maximal overhead being 3407872 bytes, while in question 4.4, when using a bitmap, the best and worst case scenario remain the same at 8192 bytes, which is much lower than the worst case scenario of the linked list, even though the best case scenario for the linked list is much lower than the best case scenario for the bitmap.

Bitmaps:

Advantages:

Time efficiency: Free memory blocks can quickly be found in bitmaps. Each bit in the bitmap represents the allocation status of a fixed-size memory block, allowing for O(1) lookup time.

Minimal storage overhead for large allocations: Bitmaps typically require less memory overhead compared to linked lists for large memory spaces. They can represent the allocation status of large contiguous memory regions with minimal overhead. This can be seen from the difference between the worst case for bitmaps being 8192B and the worst case scenario for linked lists being 3407872B.

Simple Implementation: Bitmaps are relatively simple to implement, requiring basic bitwise operations for manipulation, which can be computed faster as compared to arithmetic operations.

Disadvantages:

Wasteful for Small Allocations: Bitmaps can be wasteful for managing small memory allocations, especially when dealing with fragmentation. Each bit in the bitmap represents a fixed-size memory block, which can lead to wasted space for small allocations, which can be seen from the best case overhead for linked lists being much lower at 52B as compared to bitmaps having a best case overhead of 8192B.

Limited to Fixed-Size Blocks: Bitmaps are typically used for managing memory blocks of fixed sizes. Bitmaps offer less control over managing variable-sized memory blocks.

Linked Lists:

Advantages:

Flexibility in Memory Block Sizes: Linked lists can efficiently manage memory blocks of varying sizes, making them more suitable for dynamic memory allocation.

Less Wasteful for Small Allocations: Linked lists have less overhead in the best case scenario, or when there is only a small amount of memory allocated as compared to bitmaps, with linked lists having a much lower best case scenario of 52B as compared to that of the bitmap, which has a best case scenario of 8192B.

Disadvantages:

Higher Storage Overhead: Linked lists typically have higher storage overhead compared to bitmaps, especially for managing large memory spaces. Each node in the linked list requires additional memory for storing pointers to the next and previous nodes, which can be seen from the linked list having a much higher worst case space requirement as compared to the bitmap.

Slower Search Time: Linked lists may have slower search times compared to bitmaps, especially for finding free memory blocks in large memory spaces. Traversing the linked list to find a free block may result in O(N) time complexity in the worst case, as compared to memory access time in a bitmap being O(1).

Question 4.6 (2 mark)

Only a subset of the operations in the worst-fit allocation algorithm can be transformed from O(n) to O(1), namely the operation of finding the largest memory block in O(1) time.

We have decided on performing this using a priority queue with the underlying structure being a balanced binary search tree, which sorts the memory blocks by length,. This allows us to poll the largest free memory block in O(1) time, allowing allocation to be performed in O(1) time.

After splitting the block, we can reinsert it back into the heap, although this will take O(log n) time (which is one of the disadvantages).

Another problem with this modification is that the deallocation process would have a time complexity of O(n), as a result of mergers between adjacent free blocks having to be done by scanning the entire priority queue for adjacent free nodes, which would take more time as the number of entries in the priority queue increases.

**TOTAL: \_\_\_\_\_\_\_\_\_\_\_ / 10**